

Software-Based Self Test for Embedded Processors Enhanced by Structural Information

Tobias Koal, Christian Galke, Heinrich T. Vierhaus
Computer Science Department

Abstract

Embedded processors are often used in systems that are both safety-critical and long-living. Therefore such devices need to be tested not only after production, but also in the field of application. Then both the amount of test data and the duration of a test process are critical, since a high-quality test during the start-up phase of the overall system is strictly limited in time. Logic built-in self test using the scan chains that are implemented for production test is one option. An alternative solution that uses a functional test approach, which is enhanced by structural information, is proposed. Research in this area is performed within the scope of the DEDIS (Dependable Embedded and Distributed HW/SW-Systems Based on Nanoelectronics) class of Cottbus International Graduate School.

Kurzfassung

Eingebettete Prozessoren arbeiten heute in großer Zahl in Systemen für die Steuerung und Regelung von Maschinen, Anlagen, Fahr- und Flugzeugen und damit in sicherheitskritischen Anwendungen. Für hoch-zuverlässige Systeme sind dann Funktionen des Selbsttests notwendig. Diese müssen in kürzester Zeit und mit minimalem Leistungsverbrauch ablaufen, sollen aber alle möglichen Fehler der Hardware überdecken. Nachfolgend wird dazu ein Verfahren präsentiert, das einen mit Strukturinformationen über den inneren Aufbau des Prozessors angereicherten funktionalen Test verwendet. Themen dieser Art werden als ein Schwerpunkt in der Klasse DEDIS-Nano (Zuverlässige eingebettete und verteilte HW/SW-Systeme basierend auf Nano-Strukturen) der „Cottbus International Graduate School“ bearbeitet.

1 Introduction

Production test technology for general purpose processors has experienced a rapid development over the last 10 years. Since pure functional tests cannot even be close to exhaustive within a reasonable time frame, scan-based structure-oriented test technology has become indispensable [1,2]. Still software-based self test technology (SB-ST) has also experienced some development, since savings on test circuitry can be substantial [3,4,5,8-11]. Essentially, such technologies have three advantages over structure-oriented scan tests.

First, the test process is performed in a mode of real operation. Therefore test-induced faults that may occur during scan test only because of excessive strain on VDD- and GND rails are avoided.

Second, the sometimes excessive power consumption sometimes associated with scan test [6,9] is avoided. Third, such tests can be run during production test as well as “in the field”, for example during system start-up procedures. We can generally distinguish between three basic approaches. One such approach includes a close-to exhaustive search for an optimized functional test set [8], which has shown to be feasible with remarkably good results for relatively simple 8-bit processors. The second approach uses random-generated test data for functional testing [10]. The third and most promising approach extracts constraints from a processor design, which are fed to a structural ATPG tool. Then this tool, such as Mentor’s FASTSCAN [12], can generate test patterns for embedded functional blocks of a processor. Finally, this patterns information has compiled into a test program [3,4,11]. While such an approach looks promising at first, the extraction of constraints for structural ATPG and the fitting of structural test patterns into a test program have shown to be non-trivial. Results obtained with such methods, however, often do not exhibit a clear methodology, comparable results and conceptual limitations for realistic designs. In previous work, we designed a 16-bit RISC processor that was optimized to support software-based test functions in embedded systems. It was found that a relatively compact self-test program could be written that covers all micro instruction, all bus coupling faults and most static logic faults with a very low overhead in run-time and code size. Using this design, the work presented here differs from previous results in several ways. First, we used a close-to real life 16 bit RISC processor design, whose only essential drawback is the missing interrupt logic.

Second, for comparison with a semi-automated test synthesis approach, we can use a hand-drafted and optimized test program.

Third, we can compare the resulting fault coverage values with a full scan test, both with respect to static stuck-at faults and for dynamic transition faults, obtained by broadside testing [7]. To our best knowledge, such a comprehensive analysis has not been published before. The paper is structured as follows. In the second chapter, the basic approach towards constraint-controlled ATPG by constraint extraction from the architecture and the instruction set is described. In the third section, we discuss existing constraints for the test of embedded cores, and their formal handling. The fourth chapter describes the creation of templates for test routines and the “filling” process with structural patterns

is described. The fifth chapter shows how the process is performed for the SBST of embedded functional blocks such as an ALU and a floating point unit (FPU). The sixth section then presents comparative results for the whole processor structure. In the 7th section, the limitations of the approach are indicated, and forthcoming extensions are described.

2 Overall Design Flow

The basic approach followed also by previous publications in this field is to use structural properties of a processor design for optimized functional testing. The essential steps of the test composition process are depicted in fig. 1.

The micro-architecture of the processor is one starting point for identification of functional units that may undergo an “embedded” structural tests. Mainly combinational units such as the ALU, a floating point unit (FPU) or a shifter are prime candidates. However, also functional units of limited sequential depth such as pipelined arithmetic units are candidates.

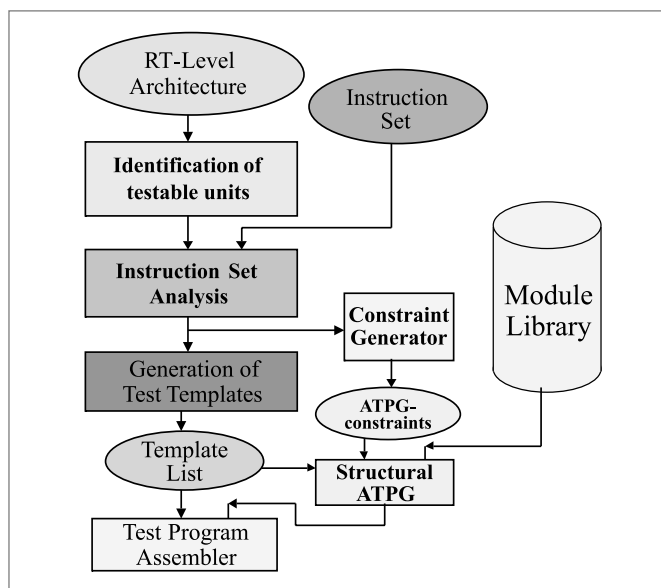


Figure 1:
Overall test composition process

In relatively simple processor architectures, this step can be performed manually. The next step is to identify instructions from the machine instruction set that are required or can favorably be used to perform tests on such identified units. Essentially, this analysis will identify all machine instructions that may serve test purposes for the respective functional unit. In order to validate control structures, all machine instructions to be performed using such a unit will have to be executed at least once. For sufficient fault coverage, instructions may have to be executed repeatedly. In earlier work, which served to create a hand-optimized test program, this analysis was performed manually by hand on the base of the processor’s micro-instructions.

The constraint generator is actually the most critical unit. It has to identify which input settings for a specific functional units, for exam-

ple the arithmetic-logic unit (ALU), cannot be applied in a functional test mode. It was found out that essentially there are two types of constraints. The first type describes input settings which cannot be applied for a 1-step static test. The second type describes constraints that exist with respect to applying a sequence of 2 patterns for dynamic test. As to be shown, it is these constraints that really reduce the functional fault coverage significantly.

After a testable unit is identified and suitable instructions have been selected, a “test template” for a respective functional unit is created. This is essentially a set of instructions, whose operands are to be filled according to the results from structural ATPG. With existing constraints for functional units, a structural test pattern generation is performed. Such a template, which is supposed to test a specific target fault (or a set of target faults) in a functional unit always consists of instructions which

- initialize source registers
- execute an instruction
- capture and store test responses.

Fortunately, commercial ATPG tools allow for constraint-driven test generation [12]. In the final step of test synthesis, a test program is assembled by merging structural test patterns and templates, for example by filling operand fields in instructions with ATPG-generated information. In the concluding step, the resulting fault coverage was identified by using a sequential fault simulator, both for static and dynamic fault coverage.

3 Testing functional units under constraints

Testing embedded functional units in a processor is more complex than expected, even for relatively simple architectures. This is already significant for static tests and becomes even more of a problem, if test pattern sequences have to be applied in order to test for dynamic faults. The basic approach taken and the essential problems for software-based BIST are shown for an arithmetic logic units (fig. 2) as an example.

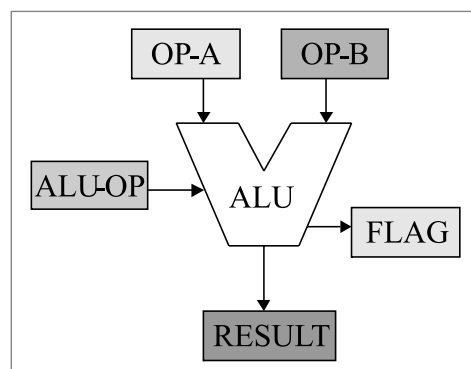


Figure 2:
ALU as an object for software-based BIST

The steps to be performed under program control are:

1. Instruction (s) to load source registers (OP-A, OP-B)
2. Loading ALU-OP-register for instruction control
3. Execute instruction and propagate ALU-results (faults) to RESULT-register and FLAG-register

4. Capture results in registers before a new input to ALU inputs arrives
5. Save information in FLAG/RESULT registers before overwrite by next instruction
6. Test response evaluation

This procedure is sufficient for functional testing of static faults. However, the procedure becomes more complex, if dynamic faults are to be tested.

1. Load OP-A, OP-B and ALU-OP for initialization
2. Execute operation, no storage of result
3. Re-load source register(s) with operands
4. Capture test response in RESULT, FLAG registers
5. Save test output
6. Test output evaluation

The problem here is that any re-loading of a source register may trigger an ALU operation. For example, re-loading OP-A may directly change the internal status of the ALU, although the ALU is not explicitly used and the next operation is a re-loading of OP-B. In this case, internal ALU conditions may be modified, only the result is not captured in a register. Therefore two load operations followed by an ALU operation will not deliver a valid dynamic test. Such features have to be “known” to a test pattern generation process.

A further analysis of the processor design exhibits that even in case of ALU instructions that seem to depend on one input only the other inputs may play a role. For example, an ALU operation (“MOVE”) that copies the content of a register A to a register B may essentially use both ALU inputs performing an operation of $OUTPUT = OP-A + OP-B$ with $OP-B = 0$. The constraints associated with operands in the ALU registers are shown in fig. 3.

Operation	ALU-Opcode	OP-A	OP-B
add [RESULT], [OP-A], [OP-B]	000	<00h-FFh>	<00h-FFh>
inc [RESULT], [OP-A]	000	<00h-FFh>	<01h>
sub [RESULT], [OP-A], [OP-B]	001	<00h-FFh>	<00h-FFh>
and [RESULT], [OP-A], [OP-B]	010	<00h-FFh>	<00h-FFh>
or [RESULT], [OP-A], [OP-B]	011	<00h-FFh>	<00h-FFh>
mov [RESULT], [OP-A]	100	<00h-FFh>	<00h>

Figure 3:
ALU instructions and allowed contents of associated registers

The third related problem is that ALU control inputs may contain “don’t care” values. Depending on the processor circuit design, the control inputs of the ALU for a clock cycle when it is not needed (and the RESULT register is not enabled for capturing) may have arbitrary inputs. However, the internal transitions of the ALU will depend on these functionally irrelevant settings. If and in what sense functionally irrelevant control signals are set, however, influences the testability of specific transitions within the unit under test significantly. As such information is not available from the instruction set, it has to be derived from a list of micro-operations associated with each instruction or, if not available, from experimental running of software. While the ALU, due to relatively many inputs for functional selection that are

difficult to control, is a close-to worst case example, also blocks such as floating point units (FPUs) suffer from the same kind of problems.

The essential problem for SBST is to extract such non-trivial constraints from a processor design, encode them as “constraints” and use them to control a test pattern generation tool. If the internal architecture including the control bits is not known in detail, the information on specific bits in control registers that have to be set or must not be set can be derived from observation of the processor model while running existing code. The essential problem associated with such a procedure is shown in fig. 4.

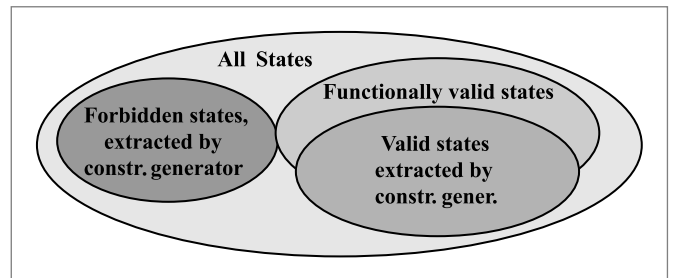


Figure 4:
State space coverage by constraint extraction

Assuming that a processor state is determined by the settings of registers and flip-flop bits in a specific clock cycle, only a sub-set of all states can be reached functionally. The constraint extractor is supposed to identify control bit settings that never occur on one side and settings that are needed for functional tests on the other hand. In reality, the constraint extractor should at least identify all control bit settings that lead to non-functional states. Even this objective will not be reached with a limited number of example instructions. On the other hand, not all control bits settings that are favorably used for functional testing will be found. This seems to be the essential bottleneck that limits the functional testability of complex processor architectures. The type of constraints that can be used to control a structural ATPG tool is shown in fig. 5. Settings of control registers such as ALU-OP can be used to restrict the contents of data registers.

Register	States & Dependencies
[ALU-OP]	<000,001,010,011,100>
[OP-A]	<00h-FFh>
[OP-B]	<00h>, if [ALU-OP] equ <100> <00h-FFh>, if [ALU-OP] nequ <100>

Figure 5:
ATPG constraints on registers

4 Test template generation

If a functional unit is to be tested for a specific target fault, source registers have to be loaded with suitable operands first. Next, a suitable instruction that triggers an operation of the functional unit is performed. Finally, test outputs are secured (fig. 6).

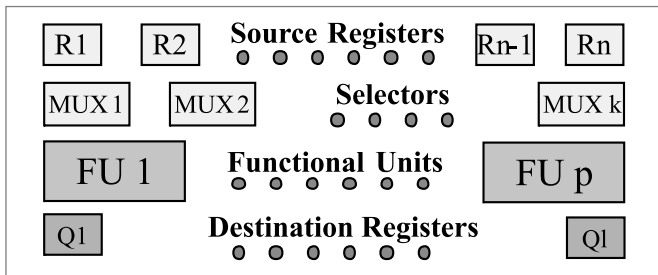


Figure 6:
Setting frames for tests of functional units

Therefore, a template test program for an embedded functional unit has to be composed, based on available machine instructions. In our case, the assembly of the template was done manually. The analysis was performed in several steps by finding

- instruction(s) that can initialize the inputs of a functional blocks
- instruction(s) that activate the functional block in a suitable way
- instruction(s) that can read the outputs of the functional block
- restrictions on input signals of a functional unit that are possible/impossible to set.

The analysis of a functional unit against the instruction set has to be done under specific side-conditions and restrictions for tests that are optimized to cover static faults on one hand and dynamic faults on the other side. For static faults, the test response generated upon a specific input bit vector has to be recorded directly before this output can be overwritten by a subsequent instruction. For dynamic faults, there are always two (or more) instructions, one for initializing a specific test condition and the next one to perform the test. If there is the choice, the test instruction is preferred which records the test output most directly. An example template that is used for ALU testing is shown in fig. 7.

Operation	Function
set [OP-A], {data};	load register [OP-A]
set [OP-B], {data};	load register [OP-B]
{ALU-OP} [RESULT], [OP-A], [OP-B];	ALU operation
save [RESULT], [MEM];	store register [RESULT]
save [FLAG], [MEM];	store register [FLAG]

Figure 7:
Template for ALU testing

First, the registers are loaded by specific instructions (MOVE, LOAD), then the addition is executed. Finally the output results are secured for further analysis. The template will have to be run with different operands (to be filled with ATPP output) and different ALU instructions.

5 Design Examples

In the RISC processor design, the ALU and a floating point unit (FPU) served as specific examples. The template used for ALU testing is shown in fig. 8a. It includes the CONSTANT instruction to load registers.

Instruction	Function
CONST Rx {#data}	load register Rx with 16-bit constant
CONST Ry {#data}	load register Ry with 16-bit constant
{#ALU-Opcode}	ALU operation with Rx and Ry
LOAD Ra #refadr	load reference address Ra
STORE Rc #refadr	store test response from Rc

Figure 8a:
ALU test template for static faults

This template is filled multiple times by operands generated by the ATPG tool under constraints, until the required fault coverage is obtained. The template used to test dynamic ALU faults is more complex. It needs to load 2 sets of source registers first (fig. 8b).

Instruction	Function
CONST Ra {#data}	Load Ra with 16 bit const.
CONST Rb {#data}	Load Rb with 16 bit const.
CONST Rc {#data}	Load Rc with 16 bit const.
CONST Rd {#data}	Load Rd with 16 bit const.
NOP Ra Rb	Init by 1st test pattern
{#ALU-Opcode} Rc Rd	Execution of test instruct.
LOAD Ra #refadr	load reference address Ra
STORE Rc #refadr	store test response from Rc

Figure 8b:
ALU test template for dynamic faults

The processor has a NOP operation which allows to connect specific registers to the ALU input without storing the result. One constraint introduced into the ATPG process then is that the ALU op-code for the initialization must be "000", while the second (test) pattern can use any available ALU function. The second example is a 64-bit floating point unit. It is not really a combinational unit, but it includes a 4-stage pipeline. For combinational ATPG it was re-designed to work a combinational circuit. Unlike the ALU, all input registers are fully accessible and without restrictions on operand values. The template for FPU testing is given in fig. 9a for static faults and 9b for dynamic faults.

Instruction	Function
CONST Ry {#data}	Load register Ry with 64 bit constant
CONST Rx {#data}	Load Register Rx with 64 bit constant
{#FPU - Opcode} Rx Ry	FPU operation with Rx and Ry
LOAD Ra # ref adr	Load storage addr. for reference
STORE Rx # ref adr	Store test response

Figure 9a:
Template for FPU test, static faults

The template includes the loading of source registers, execution of an FPU instruction, and storage of results for comparison. The template for dynamic tests uses 4 CONST instruction for loading two sets of source registers and a double application of the same function (ADDF, SUBF).

Instruction	Function
CONST Ra {#data}	Load Ra with 64 bit const.
CONST Rb {#data}	Load Ra with 64 bit const.
CONST Rc {#data}	Load Ra with 64 bit const.
CONST Rd {#data}	Load Ra with 64 bit const.
{#FPU-Opcode} Ra, Rb	FPU operation with Ra, Rb
{#FPU-Opcode} Rc, Rd	FPU operation with Rc, Rd
LOAD Ra #refadr	load reference address Ra
STORE Rc #refadr	store test response from Rc

Figure 9b:
Template for FPU test, dynamic faults

An analysis of the patterns generated by the ATPG tool showed a potential for further improvements. As processor data paths are regular by nature, there are optimal bit patterns to test such structures with a minimum of test vectors. The constraint-driven ATPG tool can be forced to produce such favorable patterns. Thereby the overall test length for an embedded functional unit (ALU, selector, shifter) can be reduced by 40-70% without losses of fault coverage.

6 Results

Results were calculated for the ALU and the FPU, both embedded in the processor structure, and for a total processor in a simple version without the FPU, because only for this version comparable results from a hand-crafted functional test were available. Data obtained for the ALU and for static faults show a coverage which is close to the figures to be obtained from scan test at a very low overhead in memory size and test program length.

Test by	Faults	Test. Faults	fault cov.	test cov.
full scan	657	656	99.85 %	100 %
man SBST	657	656	95.74 %	95.88 %
aut. SBST	657	656	99.54 %	99.70 %

SBST: 191 memory addresses, 433 clock cycles

Figure 10a:
ALU fault and test coverage, static faults

Remarkably, the results for constraint-controlled automatic SBST are better than an optimized hand-crafted test. They come close to scan test results. As to be expected (fig. 10b), the coverage for transition faults by functional patterns is inferior to scan test results (fig. 10b).

Test by	Faults	Test. Faults	fault cov.	test cov.
full scan	986	985	89.06 %	90.05 %
man SBST	986	985	66.23 %	66.29 %
aut. SBST	986	985	73.02 %	73.10 %

SBST: 245 memory addresses, 583 clock cycles

Figure 10b:
ALU fault and test coverage, transition faults

While, due to functionally not possible transitions, the fault coverage is only around 70 %, compared with 90% by full-scan and a broadside-test approach, the automated SBST again is superior to hand crafted functional tests. The floating point unit is a non-optimized experimental design that includes a high level of redundancy.

Test by	Faults	Test. Faults	fault cov.	test cov.
full scan	131965	53383	36.29%	89.29 %
aut. SBST	131965	53383	34.78 %	85.98 %

SBST: 17720 memory addresses, 62020 clock cycles

Figure 11a:
FPU fault and test coverage, static faults

The coverage obtained by SBIST comes close to the figures obtained by full-scan for static faults. Again, the coverage for dynamic transition faults is lower in either case.

Test by	Faults	Test. Faults	fault cov.	test cov.
full scan	131965	53383	25.89%	64.01 %
aut. SBST	131965	53383	24.91 %	61.58 %

SBST: 30989 memory addresses, 108461 clock cycles

Figure 11b:
FPU and test coverage, transition faults

Finally, the method was applied to the total 16-bit RISC processor, excluding only I/O-ports, which are functionally difficult to test without external feedback. For SBST figures, we first implemented a test procedure for the data path only. The accidental coverage for the control logic obtained is indicated. In the second experiment, the control logic received an extra treatment as a combinational block. With no direct access to internal control word bits possible, the results are observable only indirectly.

Test by	Total Faults /Cov.		Testable Faults /Cov.	
	data path	control path	data path	control path
faults	9197	3297	8880	3182
full scan	93.37%	92.87%	97.07 %	96.23 %
man. SBST	92.12%	83.01%	95.41%	86.02 %
aut. SBST for data path	92.11%	63.57%	95.39%	65.87 %
aut. SBST, enh. control	93.20 %	84.50%	96.53 %	87.55%

SBST: 4775 memory addresses, 13795 clock cycles

Figure 12:
TP 16 test coverage, static faults

The coverage obtained for testable faults comes close to full-scan values for data path faults and is remarkably high, though inferior to scan path figures, for control logic faults. The test size for scan test is about 3 times larger than the test size for SBST for static tests.

7 Summary and Conclusions

We investigated on the capabilities and limitations of a functional self test procedure for processors, which is systematically enhanced by structural information. The results obtained with respect to fault coverage come relatively close to scan test figures for static faults and are slightly inferior for dynamic faults. In either case, the effort in memory size and clock cycle count is small enough to run such tests either in the field or as part of a production test. Typical values are 10 kByte space for the test routine and a run-time of about 15 000 clock cycles for a simple 16 bit RISC processor.

This compares quite favorably with the effort needed for scan test. First results also indicate that the peak power consumption for SBST is only about 1/3 to 1/4 of scan-based testing, though the process was not really optimized in this direction. Further improvements of coverage figures may be obtained for a modified processor design which allows a direct observability of control logic outputs. With a growing demand for fault-tolerant computing and on-line fault detection for embedded processors serving as "watchdog" elements, such devices are expected to get on-line self-check capabilities. If such functions are available, the shortcomings in fault coverage specifically for control logic and dynamic faults can be compensated.

8 References

- [1] MAYBERRY, M.; JOHNSON, J.; SHARIARI, N.; TRIPP, M.: "Realizing the Benefits of Structural Test for Intel Microprocessors", Proc. IEEE Int. Test Conf. 2002, pp. 456-463, IEEE CS Press 2002
- [2] ABADIR, M.; RAINA, R.: "Design-For-Test Methodology for Motorola Power PC Microprocessors", Proc. IEEE Int. Test Conf. 1999, pp. 810-818
- [3] KRSTIC, A.; CHEN, L.; LAI, W.-C.; CHENG, K. T.; DEY, S.: "Embedded Software-Based Self-Test for Programmable Core-Based Designs", IEEE Design and Test of Computers, July/August 2002, pp. 18-27
- [4] LAI, W.-C.; CHENG, K. T.: "Instruction-Level DFT for Testing Processors and IP Cores in Systems-on-a-Chip", Proc. Design Automation Conference (DAC01), ACM Press, New York 2001, pp. 59-64
- [5] PASCHALIS, A.; GIZOPOULOS, D.; KRANITIS, N.; PSARAKIS, M.; ZORIAN, Y.: "Deterministic Software-Based Self-Testing of Embedded Processor Cores", Proc. DATE (Design, Automation and Test in Europe) 2001, pp. 92-96, Munich, 2001
- [6] WANG, S. AND GUPTA, S.: "ATPG for heat dissipation minimization during test application", IEEE Computer, vol. 7, pp. 256-262, February 1998
- [7] SAVIR, J.: "On Broadside Delay Test", Proc. 12th IEEE VLSI Test Symposium, 1994, pp. 284-290
- [8] CORNO, F.; SONZA REORDA, M.; SQUILLERO, G.; VIOLANTE, M.: "On the Test of Microprocessor IP Cores", DATE2001: IEEE Design, Automation & Test in Europe Conference, Munich (Germany), 13-16 March 2001, pp. 209-213
- [9] ZHOU, J.; WUNDERLICH, H.-J.: "Software-Based Self-Test of Processors under Power Constraints", Proc. of the 9th Conference on Design, Automation and Test in Europe (DATE), Munich, Germany, March 06-10, 2006, pp. 430-436
- [10] SHEN, J. AND ABRAHAM, J. A.: "Native mode functional test generation for Processors with application to self test and design validation", Proc. IEEE Int. Test Conf 1998, pp. 990-999
- [11] CHEN, L.; DEY, S.: "DEFUSE: A Deterministic Functional Self-Test Methodology for Processors", Proc. 18th IEEE VLSI Test Symposium, 2000, pp. 255-262
- [12] MENTOR GRAPHICS FASTSCANTM: http://www.mentor.com/products/dft/atpg_compression/fastscan/index.cfm



Tobias Koal is a master student in the "Information and Media Technology" program at BTU Cottbus. The work presented here was part of the investigations done for his master thesis in the Computer Engineering Group. He will graduate by the end of 2007.



Christian Galke received a diploma degree in computer science from BTU Cottbus in 2001. Since then he has been an assistant professor with the Computer Engineering Group of BTU. He was the supervisor of the work done by Tobias Koal. His own research work is closely related and has a focus on test and self test technologies for embedded processors.



Heinrich Theodor Vierhaus received a diploma degree in electrical engineering from Ruhr-Universität Bochum in 1975 and a doctorate in electrical engineering from the University of Siegen in 1983. He has been a full professor for computer engineering at BTU Cottbus since 1996.